2

Gestão de Memória Dinâmica

" Qualquer pessoa pode escrever um programa que um computador entende, mas os bons programadores escrevem programas que os humanos entende"

- **Martin Fowler** -

Sumário:

2.1 - Introdução

Os programas que devenvolvidos numa primeira disciplina de programação,normalmente utilizam a **gestão de memória estática**. Esses programas caracterizam-se por determinar a quantidade de memória para sua execução em tempo de compilação, como consequência, o espaço necessário para o programa ser executado não pode ser alterado e fica ao dispor do programa de forma exclusiva durante a sua execução.

Neste capítulo estudaremos metódos para desenvolver programas que utilizam a **gestão da memória de dinâmica**, ou seja, a quantidade de memória para executar um programa poderá ser aumentada ou diminuida em cada instante. Para dominar esses métodos o leitor terá a necessidade deconhecer às técnicas de manipulação de ponteiros.

2.2- Gestão de Memória Dinâmica

Para desenvolver programas que utilizam a gestão de memória dinâmica, necessitamos de funções especiais para solicitar blocos consecutivos de memória. Uma vez obtido esses blocos, o programa vai armazenar dados e, quando essa quantidade de memória não for mais necessária, esses blocos serão devolvidos a memória principal.

Se o programa solicitar blocos de memória e não os libertar e se essa practica for repetida, o programa poderá apropriar-se de forma exclusiva de uma grande quantidade de memória e bloquaer o funcionamento do sistema operativo.

2.3 - Alocação de Memória Dinâmica

Para solicitar um bloco de memória durante a execução do programa, é necessário invocar a função **malloc** *(memory allocation),* que possui a seguinte sintaxe:

```
void *malloc (size_t size);
```

esta função recebe como parâmetro o tamanho do espaço solicitado em bytes. Como resultado ela devolve um ponteiro que faz referência ao primeiro byte do bloco disponibilizado ou um ponteiro nulo, NULL, se o pedido não for satisfeito.

Por exemplo, para solicitar 1000 bytes de memória e, apontar para o endereço desse bloco de memória um ponteiro ptn, devemos utilizar as seguintes linhas de código:

```
void *ptr;
ptr = malloc(1000);
```

O ponteiro retornado pela função malloc() é um ponteiro genérico que não possui um tipo de dados específico. Por este facto, elei declarado como void *.

2.4 - Operações de Leitura e Atribuição

Com ponteiros genéricos ou ponteiros do tipo void *, não podemos efectuar operações de atribuição e de leitura, porque o programa não consegue armazenar dados nas porção de memória que foram alocadas.

Para tornar essas operações possíveis, é necessário converter de forma explicita, através do operador cast, o endereço obtido pela função malloc() para um ponteiro com o tipo de dados que iremos manipular.

Por exemplo, para armazenar um número inteiro na memória dinâmica, devemos converter o ponteiro genérico para um ponteiro do tipo inteiro.

```
void *ptr;
int *pnumero;
ptr = malloc(1000);
pnumero = (int *)ptr;
```

Agora, o ponteiro pnumero e o ponteiro ptr apontam para o início de um bloco de memória dinâmica obtido pela função malloc() e, através do ponteiro pnumero, o programa poderá manipular esse bloco de memória como um vector de números inteiros.

Por exemplo, podemos armazenar no endereço apontado por pnumero um valor qualquer como fazemos com as variáveis elementares:

```
*pnumero = 10;
```

Contudo, não temos necessidade de declarar o ponteiro genérico ptr, podemos armazenar o endereço retornado pela função malloc() num ponteiro específico.

```
int *pnumero;
pnumero = (int *)malloc(1000);
*pnumero = 10;
```

O ponteiro retornado pela função malloc() é a única forma de aceder à um bloco de memória dinâmica disponibilizada. Através desse ponteiro o programa pode escrever e ler dados nesse espaço. Mas se durante a execução do programa este ponteiro for pedido, esse espaço permanecerá alocado até o fim do programa e a sua localização será impossível de detectar.

2.5 - Libertação de Memória Dinâmica

Um bloco de memória solicitado pela função malloc() permanece sob propriedade de um programa por tempo indeterminado. Quando esse programa não precisar mais dele, é necessário devolve-lo para à memória principal. Essa acção é feita pela função free() que possui a seguinte sintaxe:

```
void free (void *ptr);
```

onde

*ptr é um ponteiro para um bloco de memória que deve ser liberado.

Uma vez liberado o bloco de memória, é impossível aceder a esse espaço e,após a execução dessa função, todos os ponteiros que fazem referencia a esse espaço tornam-se inválidos.

Constitui uma boa práctica de programação, executar a chamada de uma função free() para cada ponteiro obtido por uma função malloc(). Desta forma temos a certeza que toda a memória alocado foi libertada.

2.6 - Alocação Dinâmica de Vectores

Uma das aplicações mais interessantes na gestão de memória dinâmica é a criação e a manipulação de vectores. Nos programas que desenvolvemos, declaramos um vector com MAX elementos, onde MAX é uma constante, e esse vector é normalmente carregado com um número inferior de elementos.

Durante o processo de compilação é alocado um espaço para armazenar esse vector, mas o programa não utiliza normalmente todo o espaço disponibilizado. Uma parte desse espaço é desperdicado e só volta a estar disponível para o computador, quando o programa terminar a sua execução.

O nosso objectivo é declarar o tamanho do vector a maior precisão, ou seja, o vector vai ser criado em tempo de execução com o tamanho exacto para que não se despedice nenhum bloco de memória.

Para atingir esse objectivo, o número de elementos que serão armazados no vector, não pode ser uma constante, mas uma variável cujo valor será digitado pelo utilizador.

Mas, para alocarmos um vector de forma dinâmica, o compilador calcula o número de bytes que vai necessitar e esse número depende da arquitectura do computador. Por exemplo, para computadores de 16 bits um inteiro é armazenado em dois bytes, mas se a arquitectura for de 32 bits um inteiro é armazenado em quatro bytes.

Para tornar o programa genérico e o mais independente da arquitectura do computador, devemos utilizar a função biblioteca sizeof(), que possui a seguinte sintaxe:

```
int sizeof  nome da variável;
int sizeof ( tipo de dados);
```

onde

o tipo de dados é qualquer tipo de dados suportado pela linguagem C.

Esta função retorna o número de bytes necessários para armazenar esse tipo de dados ou essa variável.

Então, para criar um ponteiro para um vector, devemos seguir a seguinte sequencia de passos:

1º- Calcular coma função sizeof() o número de bytes para armazenar a estrutura;

2º- Invocar a função malloc() para retornar um apontador genérico que aponta para o primeiro byte desse bloco de memória;

3º- Converter o ponteiro genérico retornado pela função malloc() para um ponteiro cujo tipo de dados é compatível com o tipo do vector.

Mas, por consistência de cógido devemos verificar se o ponteiro retornado pela função malloc() é válido ou não. Se ele for igual a NULL , o pedido não foi satisfeito porque o computador não possui memória disponível. Para esse caso o ponteiro não é válido. No caso contrário ele é válido.

Vamos consolidar este conceito com um programa que calcula a média e a variância para n números reias.

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
   int i, n;
   float *v;
   float med, var;

   scanf ("%d", &n);                    // lê o número de elementos

    v = (float*) malloc( sizeof(float)*n ); // Solicita um bloco de memória
    if (v == NULL)
      {
        printf ("Erro: Nao ha memória suficiente \n ");
        return 1;
      }
```

```
    for ( i = 0; i < n; i++ )              // carrega os n elementos no vector
        scanf ( "%f", &v[i] );

    med = media(n,v);                      // calcula a média
    var = variancia(n,v,med);              // calcula a mediana

    printf ("Media = %f Variancia = %f \n", med, var);

    free(v);                               // liberta  os blocos de memoria
    return 0;
  }
```

onde a funções para calcular a média recebe como parâmetro o número de elementos do vector e um ponteiro para o primeiro elemento do vector.

```
float media (int n, float *v)
{
   int i;
   float s = 0.0;
   for (i = 0; i < n; i++)
       s += v[i];
   return s/n;
}
```

e a função para calcular a variância recebe como parâmetro o número de elementos do vector, um ponteiro para o primeiro elemento do vector e o valor da média.

```
float variancia (int n, float *v, float m)
{
   int i;
   float s = 0.0;
   for (i = 0; i < n; i++)
        s += (v[i] - m) * (v[i] - m);
    return s/n;
}
```

1.9 - Alocação Dinâmica de Estruturas

O procedimento para alocação dinâmica de estruturas (struct) é análogo ao procedimento para a alocação dinâmica de vectores. Para mostrar essa analogia, vamos a declaração de um ponto no plano cartesiano.

```
typedef struct
{
   int ordenada;
   int absissa;
 } TPlano;
```

Para criar um ponteiro para uma estrutura devemos utilizar os mesmos passos que utilizamos para criar um ponteiro para um vector, cujas linhas de código são descritas a seguir:

```
TPlano *pCoordenda = (TPlano*)malloc ( sizeof (TPlano) )
if ( pCoordenada == NULL )
  {
    printf (" Erro: Nao ha memória suficiente \n ");
    return;
  }
```

Suponhamos que pretendemos zerar todos os elementos de um vector do tipo plano. Como o vector será criado em tempo de execução, o utilizador terá de definir o seu número de elementos e esse número de elementos será armazenado numa variável n.

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
    int i, j, n;

    typedef struct
    {
        int ordenada;
        int absissa;
     } TPlano;

    scanf (“%d”, &n);                  // lê o número de elementos do vector

    TPlano  *tabPontos  = (TPonto *) malloc( sizeof(TPonto)*n );
     if ( tabPonto == NULL)
         {
            printf (" Erro: Nao ha memória suficiente \n ");
            return 1;
         }

     for ( i = 0; i < n; i++ )              // Zera os elementos do vector
       {
          (tabPonto+i)->ordenada = 0;
          (tabPonto+i)->absissa = 0 ;
        }
     free (tabPontos);
     return 0;
}
```

Não é por demais salientar que a notação (tabPonto+i)->ordenada é equivalente a notação tabPonto[i]->ordenada. A primeira está na notação de ponteiros enquanto a segunda na notação vectorial.

Sempre que um endereço de memória for alocado e não terminado ocorre um "*memory leak"* . Estes erros devem ser evitado porque esse espaço perdido numa será recuperado e, é uma fonte de entrada de vírus.